Framework for Development and Distribution of Hardware Acceleration

David B. Thomas and Wayne Luk

Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, England

ABSTRACT

This paper describes a framework, called IGOL, for developing reconfigurable data processing applications. While IGOL was originally designed to target imaging and graphics systems, its structure is sufficiently general to support a broad range of applications. IGOL provides a layered architecture, separating hardware and software components in both development and execution. Hardware developers can use IGOL as an instance testbed for verification and benchmarking, as well as for distribution. Software application developers can use IGOL to discover hardware accelerated data processors, and to access them in a transparent, non-hardware specific manner. IGOL provides extensive support for the RC1000-PP board via the Handel-C language, and a wide selection of image processing filters have been developed. IGOL also supplies plug-ins to enable such filters to be incorporated in popular applications such as Premiere, Winamp, VirtualDub and DirectShow. Moreover, IGOL allows the automatic use of multiple cards to accelerate an application, demonstrated using DirectShow. To allow transparent acceleration without sacrificing performance, a three-tiered COM (Component Object Model) API has been designed and implemented. This API provides a well-defined and extensible interface which facilitates the development of hardware data processors that can accelerate multiple applications.

Keywords: reconfigurable hardware, FPGA, hardware acceleration, Component Object Model

1. INTRODUCTION

Many researchers have begun to recognise the potential of reconfigurable hardware such as Field-Programmable Gate Arrays (FPGAs) in accelerating software applications. In particular, previous work has demonstrated the effectiveness of exploiting standard application programming interfaces (APIs) for such acceleration; these APIs include the use of plug-ins for Photoshop[8] or Premiere,[3] or OpenGL routines for graphics.[9] The main benefits of this approach include: the interface abstraction which simplifies the integration of hardware functions into a non-trivial software application; the clear separation into software applications and hardware building blocks, enabling the two to be developed in parallel; the well-defined nature of APIs facilitating reuse of the building blocks; and the opportunity of exploiting the reconfigurability of the hardware to speedup multiple plug-ins.

While the approach for acclerating API functions is appealing, each API acceleration project implements its own independent bridge from software API to hardware. Implementing this bridge is time consuming and costly. Yet there are many tasks at this level which are common to all API acceleration projects, as we shall described later. IGOL standardises facilities for these tasks in a layered approach, thereby reducing redundant functionality between API acceleration projects.

Moreover, we have identified additional issues that current API acceleration projects do not cover well. First, existing systems often do not include much run-time support for characterising the behaviour and performance of hardware implementations. Second, despite a few notable exceptions such as the JHDL system,[1] the hardware interfaces to the simulator and to the run-time environment in many systems are different. Third, there appears to be little research on systematic ways of selecting, customising and distributing hardware configurations as application building blocks.

The purpose of this paper is to show how the IGOL framework addresses the above issues. IGOL (Imaging and Graphics Operator Libraries) provides an infrastructure for software application developers to benefit from hardware accelerators supplied by expert electronic designers. The framework provides a combination of tools and libraries that offer the following facilities:

- Rapid characterisation of the behaviour and performance of hardware configurations using supplied test benches.

- Automatic hardware discovery and utilization. The framework is capable of discovering hardware components on the host and machines in a network, and automatically selecting appropriate ones.
- Integrated simulation support. The same code can be used for simulation, testing and production.
- High performance plug-ins for real-world applications. Hardware configurations can be packaged and deployed for supporting applications without writing a plug-in; the packaging also facilitates its distribution in either source or compiled format, possibly encrypted or compressed.
- Automatic module discovery and installation. New acceleration modules and updates can be automatically discovered and installed during application lifetime.
- Built-in support for macro pipelining of data packets. Video frames can be processed in a pipelined manner taking advantage of threading and increasing hardware I/O utilization.

IGOL is based upon the Microsoft Component Object Model (COM) framework[7] for component based development. This approach has two benefits: provision of a standard environment familiar to existing application developers, and the possibility of harnessing existing COM-based applications. In contrast, systems involving the development of a new infrastructure, such as tools built using Java,[4] may not enjoy such benefits so readily.

2. OVERVIEW OF APPROACH

An important goal of a software-hardware support framework is to enable developers to focus on their area of expertise. Software application developers should not need to worry about what hardware is currently in operation, or how the hardware configuration is found. Conversely hardware configuration developers should not be required to write much software, and they should be isolated as much as possible from platform-specific details, such as communication modes supported by the host.

To protect developers from unnecessary details, it is critical that well-defined interfaces are available which provide as much abstraction as possible without sacrificing performance and generality. To attempt this with just one interface would be impossible, as the two types of developers work in different domains, one with software and the other with hardware. Instead we use a layered approach, defining interfaces appropriate to each domain. This means that at least two separate interfaces are needed to achieve an acceptable solution from both points of view. Between these two interfaces, a combination of compile-time and run-time techniques are needed to close the gap.

The IGOL framework is intended to take over as many common hardware-software interfacing tasks as possible, such as:

- discovering available hardware resources at run time,
- determining whether hardware components found meet the requirement,
- transferring configurations to hardware,
- synchronizing with the hardware control logic,
- moving input data and results between host and hardware.

So far we have focused on having a simple way to talk to hardware. However, even with a Hardware Abstraction Layer for communicating with devices, there is too much coupling between the software applications and the hardware implementations. What can be useful is a task-orientated interface, with no reference to hardware resources or configurations. Rather than asking for a hardware accelerated operation for a particular piece of hardware, the application would be able to just request the operation itself and let the framework determine the best way of implementing that operation, given appropriate resources. A virtualized view of operations provides several benefits for both application developers and hardware developers, for instance:

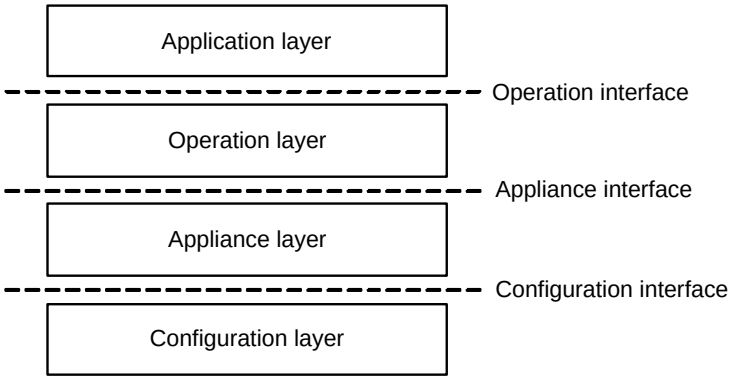

Figure 1. Overview of framework.

- Complete hardware independence as applications are bound to an abstract operation rather than a particular hardware appliance or configuration.
- Applications can fall back to software implementations if hardware is unavailable.
- Behind the abstract interface, an operation can be located on another machine where the right kind of hardware acceleration can be found.

By using a layered approach (Figure 1), we can create a framework with little coupling between applications and hardware configurations. This framework provides a richer environment for applications than if they were bound directly to the hardware implementation. An operation interface provides many opportunities for run-time discovery and optimization of acceleration without affecting client applications. Providing an abstraction layer over the hardware allows developers to use the same code for simulation and for hardware execution. Moreover, it simplifies the access to hardware on remote machines for testing.

The IGOL framework is an implementation of a layered API and run-time system as outlined above, allowing easy development, testing and deployment of hardware acceleration. Acceleration is exposed to applications through a simple while generic interface which supports a wide range of data-processing tasks. The framework also supports hardware development using the Handel-C language and the RC1000-PP system[2] using a simple appliance layer (Figure 1).

The following provides details of our implementation of this approach using Microsoft's Component Object Model (COM)[7] as the main interface medium. COM is an enterprise class distributed object system, providing support for the use of software components through well-defined interfaces. We build on the strong separation between interface structure and implementation that COM encourages, to provide a system with interfaces that are both structurally and functionally well-defined. This allows us to select the actual implementation at run time for optimal performance.

3. DESIGN TOOL STRUCTURE

This section gives a more concrete description of the framework's organisation (Figure 2), explaining the interactions between application, library and hardware components. The use of several terms as they relate to the system should first be explained:

- Operation: an abstract data processor that implements a defined function.
- Operation instance: a concrete operation acquired at run time.
- Module: a packaged unit, which contains one or more operation implementations and associated meta-data.

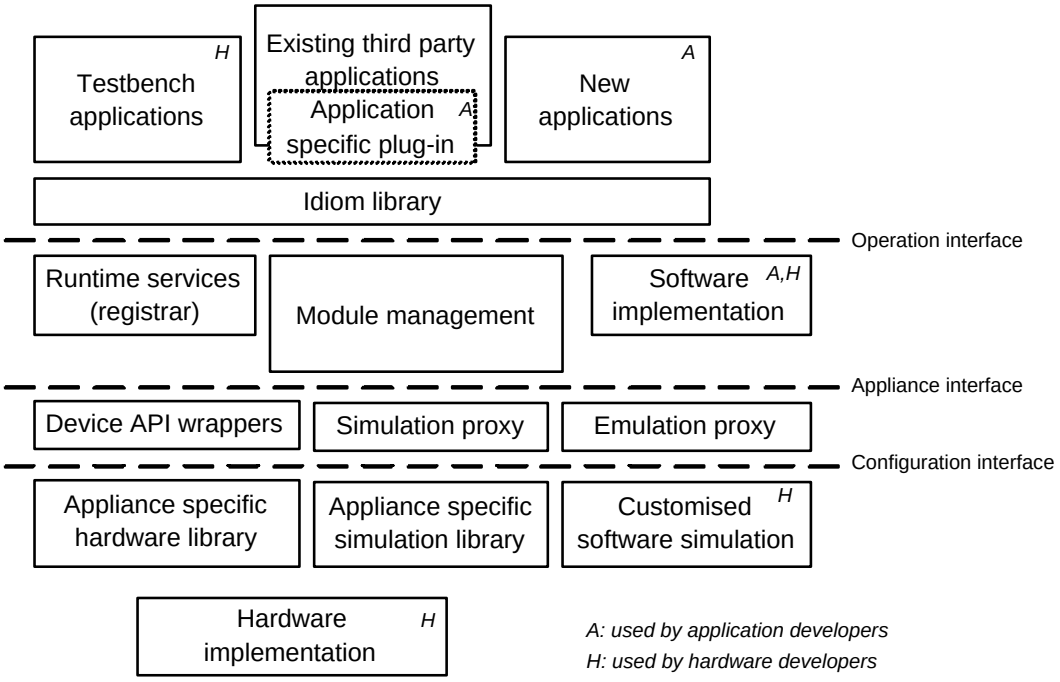

Figure 2. A more detailed overview of IGOL implementation, showing the modules of interest to application developers and hardware developers. A description of the components in this framework can be found in Section 3.

3.1. Application layer

The application layer is mainly responsible for data production and consumption. From the application layer, the only visible interfaces are those exposed by operation instances.

The framework provides a number of applications that serve as a testbench during the testing and debugging of implementations, both hardware and software. These can exploit debugging interfaces to provide capabilities that normal applications do not provide, such as:

- automatic logging of steps performed and any failures;
- conformance testing of new implementations against reference implementation;
- automated benchmarking and performance testing;
- browsing of available operations and modules.

The test applications can also be used as lightweight demonstration platforms, or even as final host applications for very simple tasks. For instance, the command line test application can be used to send data through hardware using `stdin` and `stdout` channels.

Hardware used to acceleratate a certain operation for a third party application may potentially be used in other applications. An accelerated video effect could potentially be of use in many different situations, from video editing suites such as Premiere to streaming frameworks such as DirectShow. Unfortunately the plug-in frameworks vary tremendously in the details of how a plug-in is discovered, initialised and accessed, even though the core tasks performed are the same.

If a generic IGOL compatible plug-in is created, all available operations can be exposed through it. At initialisation time the plug-in can query the registrar to determine all the operations that can work within the particular applications constraints. For example Premiere uses three classes of plugins for different types of media and functions, each with their own interface:

- Video Effect: video $\longrightarrow$ video
- Video Transition: 2 $\times$ video $\longrightarrow$ video
- Audio Effect: audio $\longrightarrow$ audio

The IGOL framework provides a generic plug-in for each type that determines the set of compatible operations, and makes them available to Premiere.

In-house applications can also be easily adapted to use IGOL directly. As with developing a plug-in, this is something that only needs to be done once for each type of operation. This is made easier by the use of the Idiom layer (Figure 2), which provides a thin wrapper layer over the raw operation interfaces. This simplifies common usage patterns of the registrar and operation instances.

3.2. Operation layer

The operation layer is concerned with the management and implementation of operation instances for use by applications. The main elements of the layer are the registrar, surrogate operation instances for hardware configurations, and software implementations.

The registrar is the collective name for all the framework components that are involved with collating and processing the meta-data associated with modules and the operations they implement. The most important tasks the registrar performs are:

- Module discovery: collecting information about modules as they are installed;
- Operation mapping: mapping an application's request for an abstract operation to concrete implementations in the available modules;
- Operation selection: determining the the best implementation of an operation given the resources available.

When an application asks the registrar for an operation, it receives an operation instance back, assuming its request can be met. How this instance is implemented depends on the meta-data that the module exposed to the registrar.

A module can contain software which forms a complete implementation of the operation instance interface. In this case, the registrar simply delegates to the module's implementation. This allows advanced or extended modules to retain control of their interface at the expense of implementing everything from scratch.

Alternatively a more generic shell module can be created using a wizard in the framework support tools. This provides a default implementation of the operation interface which can be extended as necessary. This is particularly useful for creating software implementations, as an existing implementation of an algorithm can be packaged quickly and easily.

When a module contains a hardware configuration, the registrar has to provide a surrogate operation instance. For each defined configuration interface or communication protocol, there is a surrogate module that understands the protocol and can manage the synchronisation and data transfer over that protocol. The registrar determines which surrogate to use based on meta-data included with the hardware configuration in its module. If a standard hardware surrogate is not acceptable, then a custom surrogate can be developed and distributed either in the same module, or as a separate shared module for use with multiple hardware configurations.

Hardware surrogates are largely hardware independent, and only communicate using the appliance interfaces. The appliance interface is merely the medium: the surrogates communicate with the configuration, via the appliance.

3.3. Appliance layer

As explained above, the appliance layer is simply a communications channel between software and hardware. The aim is to make all hardware appliances look the same, whether they are real or simulated.

The appliance interface is designed to support high-level abstractions of the underlying hardware without incurring significant performance overheads. This means that the interface must be fairly device specific. The only platform supported by IGOL currently is the RC1000-PP system,[2] and the corresponding appliance interface reflects the features of this card for performance. This does not mean that it is specific to the RC1000-PP; rather it is specific to this class of devices. A different style of applicance, such as the Pilchard,[5] would need a different appliance interface to reflect its different I/O characteristics.

As well as the lightweight wrapper of the real hardware for production use, IGOL also provides some appliance interface implementations for testing and debugging hardware configurations.

- Simulation proxy: allows Handel-C simulations to be executed in a replica of the RC1000-PP hardware environment. This can be used to debug hardware configurations, or if no real hardware is available for testing.
- Software emulation proxy: supports the use of a software configuration in an RC1000-PP environment. Useful for testing complicated communications protocols when the simulated hardware is too slow.
- Network proxy: in addition to the default DCOM remoting strategy, IGOL also supplies an optimized version which provides asynchronous, pipelined operation. This reduces the effect of latency and can make remote hardware a feasible solution for operations that are heavily compute-bound, but with only medium I/O requirements.

3.4. Configuration layer

The configuration layer contains the hardware configurations that perform the accelerated operations on a given appliance. These configurations may be true hardware bit-files, or simulations of hardware; all that matters is that they can be loaded by one of the appliance layer implementations.

A configuration needs not be developed in a particular language, as long as it can use the hardware appliance's communication and data channels to implement the correct protocol. Currently the only language with built-in support is Handel-C.

The purpose of this hardware library is twofold. First, it implements communications with the hardware surrogate. This reduces the work needed to develop working hardware and allows different protocols to be tested. Second, the library provides an adaptation layer over the device specific organisation, abstracting detail like bank access and communication. The adaptation layer is very low-level, since it provides just enough separation between the device and the developer's code to support a reasonable degree of portability.

The simulation version of the library allows the developer to create a simulation of the hardware which can be packaged and deployed just as a true hardware configuration can. By default the registrar would not expose simulated hardware to standard applications, but if explicitly enabled, the simulation will be available to all clients requesting the operation. This can be used to provide an initial design verification against a reference implementation before compilation to a hardware version.

3.5. Media-types

The framework should be extensible, not just through new operations and implementations, but also through completely different types of applications and data types. To support this degree of extensibility, IGOL uses a media-type system adapted and generalised from Microsoft's DirectShow library. A media-type completely describes the format and layout of the data that will be processed using three fields:

- Major-type: indicates the broad class of data, such as Video, Audio or even just raw binary.
- Minor-type: the specific format within the major-type, such as the specific pixel organisation for video formats.

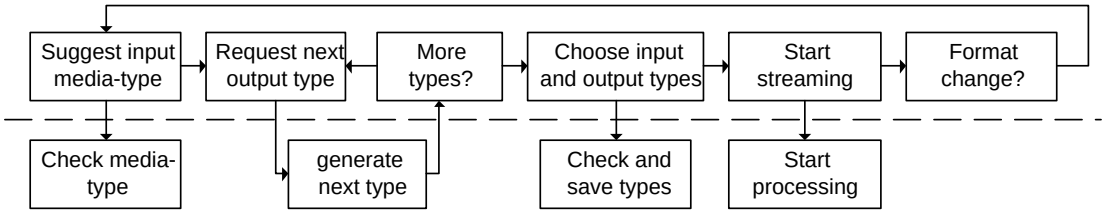

Figure 3. Media-type negotiation.

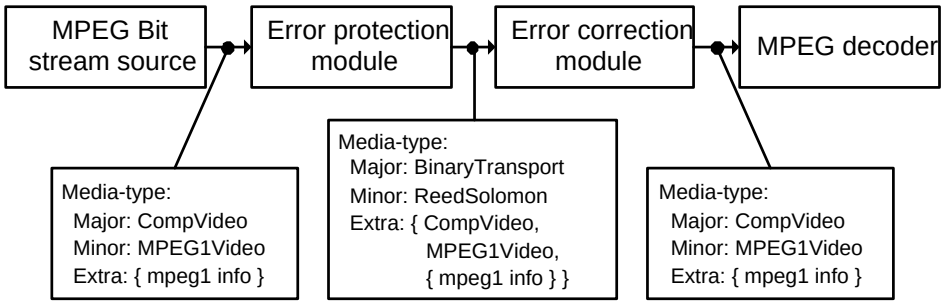

Figure 4. Propogation of media-type over error protection stream.

- Extra-data: holds any extra information needed, such as the width and height of video frames, or the frequency of audio. The structure of the data is identified through an associated id.

All the ids used in the system are 128-bit Globally Unique Identifiers (GUIDs), providing a large namespace. This means that developers can develop specialised media-types for specific uses or improved performance, while established data exchange formats are also supported. To allow applications and operator implementations to determine what supported types they have in common, the data transport interfaces incorporate type negotiation features (Figure 4). The negotiation is flexible and allows different levels of complexity, from specifying media-types directly to requesting all the types that an operation supports and choosing the quickest/highest quality.

Because these media-types are known size blocks, they can also be packaged inside other media-types. This allows media-types to be nested and propagated through intermediate types. For example, when developing a Reed-Solomon encoder/decoder pair, the incoming media-type can be encoded into the outgoing media-type by the encoder. An advanced client such as the generic IGOL plug-in for DirectShow is then able to automatically connect a compatible decoder which will expose the original media-type. This type of feature is very useful for testing and demonstrating such systems. For instance a random error generator can be placed between the coder and decoder in order to progressively add more noise until the error correction breaks down.

4. CASE STUDY

In order to show the framework in action, this section provides an example of using the system from both application developers' point of view and hardware developers' point of view. We assume a C/C++ software application that uses edge detection on every frame, which causes a performance bottleneck. Note that in this case the work is accomplished by two developers, as new hardware is being developed to accelerate a specific operation, requiring both hardware and software support.

First the operation to be accelerated needs to be defined more precisely: is the edge detection on all channels or just luma? Is thresholding performed? What particular type of edge detection kernel? Some of these decisions can be deferred by making them run-time parameters, while others would need to be fixed at compile time, such as the type of kernel

used. The aim of this step is to assign an operation id (a GUID) to the function `edge-detection` as the application sees it. If, after defining the operation precisely, a compatible operation that has previously been defined is found, then the existing id can be re-used; otherwise a new id needs to be defined that is associated with the operation definition. So far this step of defining operations and comparing them with existing definitions must be done manually, as there is no standard way of specifying or testing these descriptions against each other.

If an existing operation id can be matched, then there is always the possibility that hardware acceleration is already available and can be used straight away. The same performance bottleneck is often discovered in many applications, with the result that somebody has implemented a hardware version before. However, usually this previous version is only available in hardware description source code form, requiring the code to be read, understood, compiled and integrated into the current application. An IGOL module will already be available in a binary ready-to-run form, encouraging code re-use.

This analysis step results in an operation id, `OPID_5x5SobelEdgeDetection`, with two parameters `all_channels` to determine whether detection is performed on all channels, and `threshold_level` to determine whether or not thresholding is applied.

4.1. Application developer

The application developer needs to add three main sections to his code: discovery, setup and processing. The discovery step uses the IGOL registrar to determine at run time whether it is possible to implement a given operation using the resources on the local machine. As well as an operation id, the application needs to tell the registrar any special requirements it has, including any aspects of the operation that would not be needed. In this case the application might specify that it needs to process 640 by 480 pixel frames at 15 frames per second, and it also indicates that it will not vary one of the operation parameters after creation. These hints can either be absolute requirements, or just targets.

```
// Create an opspec instance based on our op id that we can accumulate
// info in CMLOpSpec opSpec(OPID_5x5SobelEdgeDetection);

// specify the all_channels parameter and that it wont be changed
opSpec->setParam(all_channels, false);

// specific that we would like a particular frame rate
opSpec->addHint(HINTID_Video, {640x480,15fps} );
```

This information is sufficient for the registrar to locate and initialize an implementation that can meet the requirements:

```
CMLTransform op;
try{
    op=registrar.createOperationInstance(pSpec);
    // could also have just passed the op id
}catch(CMLRegistrarError &e){
    // Deal with the fact that nothing could be found
}
```

By treating the failure to find an implementation as an exception, the developer is forced to either handle the lack of an implementation, or let the error terminate the program. Once the operation has been retrieved, we need to perform the setup step, where any remaining parameters are set, and data formats are determined. In this application only RGB32 video is acceptable, so no format negotiation is needed:

```
op.setParam(threshhold_level, 127);

// use a library class to automatically generate media-type
```

```
op.setInputFormat(CMLVideoFormat(MEDIASUBTYPE_RGB32,640,480));
op.setOutputFormat(CMLVideoFormat(MEDIASUBTYPE_RGB32,640,480));

// Force all hardware resources, memory etc. to be allocated
op.commit();
```

The transform is now set up and ready to start processing data. Usually the operation would be to set up to perform pipelined operations if the application allowed, but this step is omitted for simplicity, leaving the transform in the default one-in, one-out, synchronous mode. To process data the application simply replaces the current processing code with a call to the operations transform function:

```
// existing function that does edge detection
void doEdgeDetect(const BYTE *pInput, BYTE *pOutput)
{
    // cut out existing code, and call transform
    op.transform(pInput, pOutput);
}
```

The application is now able to use any operation instance of `5x5SobelEdgeDetection` that can be found at run time, whether it is implemented using software or hardware. Until a hardware version is ready, the existing software implementation can be used, by wrapping it up as just another implementation of the operation. This is simply and quickly done using the module wizard provided, which will generate a stub module that the code can be pasted into. This software implementation can then be distributed with the application so that an implementation is always available, independent of the availability of a suitable hardware component.

4.2. Hardware developer

In this case the hardware developers should be able to create a working version rapidly, as a C version of the algorithm is already available. This can be copied directly into a Handel-C source file, parallelized as much as possible and integrated with the Handel-C part of the IGOL library. A specialized version of the library is provided to make the common task of processing video frames easier, meaning the only entry point that must be implemented is `ProcessImages(...)`. This macro is given the source and destination buffers, and expects the implementation of the macro to transform the source buffer onto the destination buffer.

```
// entry point macro expected by framework
macro proc ProcessImages(src,dst, width,height,pitch)
{
    RGBPEL *pSrc=getBufferPtr(src);     // lock pointers to the buffers
    RGBPEL *pDst=getBufferPtr(dst);

    // adapted C version
    for(y=0;y<height;y++){
        for(x=0;x<width;x++){
                pDst[x]= ... ;  // calculate output pixel
        }
        par{
                pSrc+=pitch;
                pDst+=pitch;
        }
    }
}
```

This macro is all that the developer needs to implement, as the framework will handle all communications with the host by default. Once this macro is implemented the developer can either try simulating the code using the IGOL test bench and the Handel-C simulator, or he can compile the code into a hardware configuration and test it using one of the default module implementations to host the configuration. If hardware is not available on the development machine, the run-time library tries to locate shared hardware on other machines and test the configuration there.

Once this first version is working, the bit-file can be compressed and packaged, giving a single file suitable for distribution with the application. The file is capable of registering itself with the registrar on any machine it is installed to make itself available to any interested parties. If the first implementation is fast, but not fast enough, the hardware developer can start work on a second version. Once this is completed and tested it can be distributed and installed side-by-side with the initial implementation, allowing the registrar to find it and use it.

As the hardware description is optimised the clock rate will rise, the cycles-per-pixel will fall, and the amount of time taken by the FPGA to process each frame will decrease. As the time to process each frame decreases, the data transfer between host and card starts to become a significant fraction of the overall hardware processing time. To increase performance further the data needs to be overlapped with the processing, requiring the hardware surrogate to pipeline the video frames. Using the framework this can be achieved simply by changing the communications protocol implemented by the library. The hardware library handles the pipelined operation, and a different surrogate will be chosen automatically by the registrar.

4.3. Experiences with IGOL

The IGOL framework has been successfully used by a number of people in diverse applications since its creation:

- The framework has been extended to support new media types and applications. Third part Applications which have been adapted using plugins include: Adobe Premiere, Adobe Photshop, Microsoft's Directshow, Nullsoft Winamp, Microsoft's DirextXMedia and VirtualDub.
- Handel-C imaging libraries: image processing libraries developed for the RC1000-PP include edge detection, variable kernel filtering, bilinearly interpolated rotation/scaling and various special effects.
- Reed-Solomon encoder/decoder: an existing implementation of the Reed-Solomon is adapted into the framework. Using the DirectShow plug-in (Figure 3) both the encoder and decoder are connected together on a two-board system, with a noise generator in between. MPEG video can be streamed across the encoder, and by steadily increasing the noise the difference with and without error correction can be clearly and interactively explored.
- Barcode extractor: a barcode detector and extractor has been developed, using the testbench facilities to debug the algorithm on static images, and further test it on video fed in from MPEG and a video capture card.

5. SUMMARY

The IGOL framework has been shown to simplify the task of accelerating non-trivial applications using reconfigurable hardware for both software application developers and hardware developers. Application developers can define an operation, provide a software implementation, then move onto other work without waiting for the hardware to be finished. Hardware developers can iteratively develop and deploy configurations after an application has been distributed without new software.

Current and future work includes refining IGOL to include optimisations such as configuration transformation,[10] memory access optimisation[11] and pipeline vectorisation,[12] and extending our approach to support descriptions other than Handel-C[6] and multi-FPGA systems.[3]

Acknowledgements

Many thanks to Matt Bowen, Stephen Chappell, Roger Gook, Henry Styles, Shay Seng and Tim Todman for their comments and assistance. The support of Celoxica Limited, UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 31409 and GR/R 55931) and Xilinx, Inc. is gratefully acknowledged.

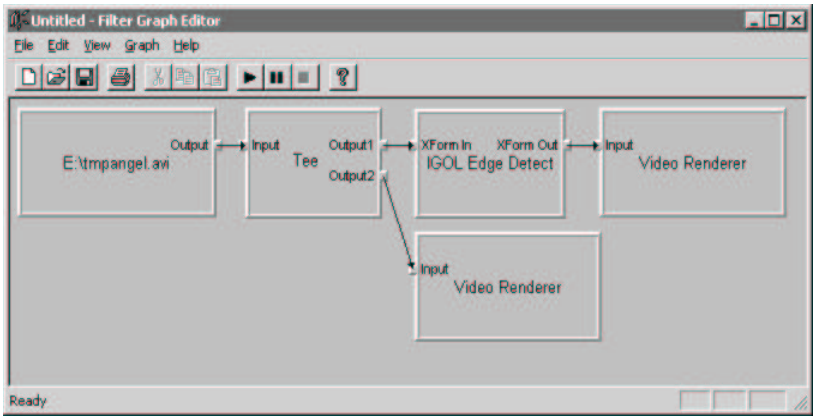

Figure 5. Using the generic IGOL plug-in in DirectShow.

REFERENCES

1. P. Bellows and B. Hutchings, "JHDL – an HDL for Reconfigurable Systems", *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1998, pp. 175–184.
2. Celoxica Limited, *Handel-C v3.1 Language Reference Manual*. http://www.celoxica.com.
3. S.D. Haynes, J. Stone, P.Y.K. Cheung and W. Luk, "Video image processing with the SONIC architecture", *IEEE Computer*, April 2000, pp. 50–57.
4. B. Hutchings et. al., "A CAD suite for high-performance FPGA design", *Proc. IEEE Symp. on Field-Prog. Custom Comput. Machines*, IEEE Computer Society Press, 1999, pp. 12–24.
5. P. Leong et. al., "Pilchard – a reconfigurable computing platform with memory slot interface", *Proc. IEEE Symp. on Field-Prog. Custom Comput. Machines*, IEEE Computer Society Press, 2001.
6. W. Luk and S. McKeever, "Pebble: a language for parametrised and reconfigurable FPGA design", *Field-Programmable Logic and Applications*, LNCS 1482, Springer, 1998.
7. D. Rogerson, *Inside COM*, Microsoft Press, 1997.
8. S. Singh and R. Slous, "Accelerating Adobe Photoshop with reconfigurable logic", *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1998, pp. 236–244.
9. H. Styles and W. Luk, "Customising graphics applications: techniques and programming interface", *Proc. IEEE Symp. on Field-Prog. Custom Comput. Machines*, IEEE Computer Society Press, 2000, pp. 77–87.
10. N. Shirazi, W. Luk and P.Y.K. Cheung, "Run-time management of dynamically reconfigurable designs", *Field-Programmable Logic and Applications*, LNCS 1482, Springer, 1998, pp. 59-68.
11. M. Weinhardt and W. Luk, "Memory access optimization for reconfigurable systems", *IEE Proc.-Comput. Digit. Tech.*, May 2001, pp. 105–112.
12. M. Weinhardt and W. Luk, "Pipeline vectorization", *IEEE Trans. Computer-Aided Design*, Feb. 2001, pp. 234–248.